RISC-V: They Should Have Known Better

dmitry.gr

Dmitry.GR

RISC-V: They Should Have Known Better

Table of Contents

1. Everything for Everyone
2. Optionality
3. Missing Obvious Pieces
4. Ridiculous encoding
5. Alleged Fixes
6. How Did We Get Here and Where to Now?
7. Does This Mean RISC-V is Doomed?
8. Comments...

I am often asked to explain my distaste for RISC-V and I often find myself explaining it piecewise. The reactions are often of the form "you just do not understand the brilliance of it all", which is, of course, no argument at all. After being asked for the Nth time to explain, I decided to put it all down in one place so that I could simply link to it when asked next. Plus, if anyone wishes, then, to form a coherent counter-argument, they could refer to my points clearly and in detail by having this text as a reference. All opinions stated here are mine and do not represent the views of my employer, any deity, or my landlord. My cats concurred in part and dissented in part and will publish their opinion later.

Everything for Everyone

RISC-V will own the cheap-as-dirt single-use microcontroller space eventually. Not due to its ISA design, but despite it.

The first and simplest-to-grasp issue is that one cannot be best for all use cases. RISC-V fans would have you believe that RISC-V will soon own all supercomputers, while also owning all the tiny microcontroller use cases, and all things in between. This is impossible, and would be equally impossible for any ISA. Simply put, the things a high-end CPU needs are diametrically opposed to the things a small cost-saving microcontroller core needs. The design choices are not merely microarchitectural, but actually (and necessarily) impact the CPU architecture itself. For what it is worth, I am 100% sure that RISC-V *will* own the cheap-as-dirt single-use microcontroller space eventually. Not due to its ISA design, but despite it. It will take this role from 8051 by being an improvement on it -- a bar so low, it is but a speed bump.

What does a cheap microcontroller core need? Let's inspect what they are used for. Typical use cases are to interface with and quickly reconfigure hardware blocks in a larger chip, eg in an MP3 player, an SD card, or a USB stick. The hard work is done by custom IP and the CPU core is just there to occasionally prod a register or configure something. What matters in this case is interrupt latency (lower is better) and size (smaller is better). Usually you would not expect much math to be done on such a core. Mass-produced cost-reduced devices would have the code running out of real ROM (if non-updateable) or RAM (if updateable); NOR flash costs too much and is not an option for really-mass-produced things. When running out of ROM, code size matters because ROMs are not very compact. When running out of RAM, code size matters because SRAMs also take up a lot of space on the die. Thus, code density matters for these use cases. Since much math is not expected, things like hardware dividers (or even multipliers) can be discarded. Privilege separation is also not needed in such single-use situations -- no external untrusted code is expected to ever be fetched. "But, " you might say, "you just described RV32IC (or RV32EC)!"

So, at basically the only purpose such an embedded core has, RISC-V is notably worse than the leading existing competitor.

Indeed, it is somewhat close, except really you need RV32I_Zicsr to claim that. Without Zicsr, there is no spec-compliant way to handle interrupts, as there is no temporary place to stash a register to allow you to stash the rest of them. MIPS reserved two kegs for this ($k0 and $k1). Without them, RISC-V needs mscratch/sscratch. Without Zicsr, you do not have those and are stuck with weird other methods to do things. And thus we are back in 8051 territory - it specializes in

doing things weirdly. Small embedded cores are not out-of-order monsters. If you get one instruction per cycle out of them, you consider yourself lucky. Given this, let's optimistically count the number of cycles needed for an interrupt handler to stash ABI-required regs and call a handler written in C. First we'll use a CSRRW to stash a reg (let's say t0 for ease of explanation) and get a base address of where we may stash the rest. Then we'll need to stash ra, sp, gp, tp, t1-t6 and a0-a7. We'll then need to use another CSSRW to get back the old t0 value and stash that as well. That's *at least* 21 cycles. On the way out, the math is similar: one CSRRW to read the address of the stashed regs, and 19 loads to load them. That's *at least* 20 cycles. But that is not all. Since this needs to be done in assembly, we'll need to actually account for the JAL to our C handler and a RET from there. We'll graciously assume those are each two cycles. Thus each interrupt has at least a 44-cycle cost before any work is done in the C handler. Cortex-M0 (the competing cheap 32-bit core) does an interrupt entry in 15 cycles, exit in 12 cycles, and since it pushes the ABI-clobbered regs in hardware, the handler is written in C directly. Thus each interrupt here has only a 27-cycle cost. Oof... that’s a lot faster! You might protest that I am being unfair by not considering RV32E here. By having half as many regs, it can do the initial push 6 cycles faster and the pop as well, bringing its interrupt overhead to 38 cycles. Still over a third more than the Cortex-M0. Oof... So, at basically the only purpose such an embedded core has, RISC-V is notably worse than the leading existing competitor. The existence of CLIC *and* various proprietary "fast IRQ" / auto-stacking extensions is an additional indictment. The base ISA forces vendors to invent non-standard silicon to reach parity with a decade-old Cortex-M0. That, in turn, further fragments the "standard" (if it can so be called). Hilariously, even with the compressed extension, the typical IRQ prologue is larger and slower than the Cortex-M0’s zero-byte hardware path.

Now, about those compressed instructions. Let us look at them in detail. They are hilariously poorly designed. Say you want to store a byte to a register plus offset. What range of offsets can a 16-bit instruction encode? Zero through three. Not thirty three, not three hundred and three. Three! Well, maybe it is better for storing a halfword? Nope... zero or two. What even? Why? At least when you store a word, you get a sane range of zero through 124 bytes, but what is going on there with those other ones? Worse, the instruction for storing a halfword is encoded similarly to the one storing a byte, but somehow it has fewer options for offsets? Why? Well, one of the bits that store-byte uses for offset is just hardwired to zero... it could have been used to expand the range to at least go to 6, but it doesn't! By comparison, Cortex-M0 is happy to let you use offsets from zero to 31 for bytes, zero to 62 for halfwords, and zero to 124 for words - clearly this covers a lot more use cases. So what happened here? Truly, I do not know, but it is indeed hard to justify. A typical refrain is to just use full-length instructions for these larger offsets. Sure, but density will suffer - the very

density that RISC-V fans were bragging about so recently when trumpeting the C extension. But wait, there is more yet. Those instructions to store a byte and a halfword are not even *in* the C extension. They are in another one called Zcb so you may not get access to them at all, even if their puny range were good enough to use in your situation. We'll get to "extensions" later...

What do server cores need? Raw throughput. Here, we are in the world of out-of-order cores where silicon is more or less free, since no matter how big your core is, the caches will dwarf it in size. Modern out-of-order cores are decoding eight or sometimes ten instructions at once, and issuing them to multiple ports concurrently; many modern cores can take more than one branch in the same cycle (think about that for a second, let it sink in ... yes). Code density is really not as much a concern here as it was in the past. It matters, but making a slightly larger L1i is not terribly complicated and, again, Si area is more or less free on the scale of such small things. What you really want is the ability to fetch and decode as many instructions at once as easily as possible. While doing that, it also helps if the instructions tell you as much about their intent as possible, to allow you to merge them with others or split them up into pieces most efficiently. Seemingly, these two desires are at odds with each other, and to some extent it is true. "Easy decoding" is, as is widely known, latin for "fixed length" while "as much as possible" is greek for "long". Obviously we do not want fixed-length very long instructions. Where do we draw the line? Having instructions be a power-of-two in length makes many other things like alignment easier, so then what? Two bytes is too short. Eight bytes is too long. The answer is fixed length 4-byte instructions are a nice middle ground. That provides enough encoding space to encode almost anything you'd want, namely: 3 registers encoded in each instr, long offsets for branches. Why does this sound familiar? Because that is what aarch64 (and A32) have proven to work exceptionally well. MIPS made the same choice for the same reason.

You might now protest that ARM also has Thumb and MIPS has microMIPS. However, in high-performance compute Thumb is dead. When Apple was designing aarch64 with ARM, much modeling and testing showed it to be a net loss for instructions per watt and instructions per second. MIPS would surely have killed microMIPS too, had MIPS lived long enough to reach the current cost-per-transistor regime. The main upshot is that compressed instructions have no business in large cores, they get in the way of fast parallel decoding of many instructions by making it slower to find boundaries. You might protest that "RISC-V makes it easy to find instr lengths", but "easy" is not the same as "instant and free" that fixed-length instructions grant you.

It took them TWO YEARS to realize that arrays exist!

Getting back to our discussion of the raw performance that high-end cores need to demonstrate. What is one of the most common operations performed by any code? Array access. This is why x86 has addressing modes of the form [ebx + esi * 4] and ARM has [R0, R1, LSL #2]. Without it, you are forced, like an idiot, to shift a register left by two, then add it to another register, and only then use that to access memory. Three instructions for a single array access. The usual excuse given for this inexcusable lack of foresight is that "instruction fusion will fuse all those three instructions into one in fast cores". Yeah... if anyone ever pulls that off, they will win many prizes. No core I am aware of fuses more than two consecutive instructions. None. The reason is quite obvious -- the combinatorial explosion of the number of possible combinations to consider, track, and handle. So now that we've established the bullshit excuse is bullshit, what is there to be done? Well, a few *YEARS* after the spec was written, an extension was proposed to help this issue -- Zba. It provides three instructions of the form SHxADD for x being 1, 2, or 3. This combines a shift with an add, basically shortening our array access from three instructions into two. This is still worse than having register + shifted register addressing mode, but at least now "the core can fuse them" becomes less bullshit and more believable, assuming someone produces such a core. One problem: SHxADD is always 4 bytes long, and the memory access instruction itself will be 2 or 4 bytes, so the array access becomes 6 or 8 bytes of code, to ARM's 4. This is where all those people who were just shouting about the wonderfulness of the C extension for density and how great it is for high-perf cores quietly shut up and look at the floor. Yeah... For extra credit, the Zba extension was only ratified in 2021, over two years after the base spec. It took them *TWO YEARS* to realize that arrays exist!

You know you really fucked up bad when you manage to make Qualcomm sound like the voice of reason.

Curiously, this would be easy to fix. Currently 3/4 of the encoding space is allocated to compressed instructions (all instructions whose lower 2 bits are not 0b11). Reusing some of that encoding space for better addressing modes is a no-brainer and would produce denser code with better array addressing ability. Unfortunately, it would make too much sense for anyone to actually do. Of all possible champions of sanity, Qualcomm ... proposed doing this, and even prototyped it. It went nowhere... And you know you really fucked up bad when you manage to make Qualcomm sound like the voice of reason. But, back to our SHxADDs. There is no guarantee that you'd get to use them anyways, since Zba is an extension and is thus optional. Are you getting tired of hearing "optional" yet? Let's talk about that next.

Optionality

What does RISC-V have in common with USB-C and RCS? These things are all ostensibly standards, sure, but the interesting part is that claiming to be in compliance with one of these standards means NOTHING while being technically true. Is my USB-C cable wired only for USB 2.0 valid? Sure, USB 3 twisted pairs are optional. Is my non-e-marked cable valid? Sure, e-markers are optional. Can my USB 3.0 USB-C cable choose to not support 20Gbps? Sure, 20 Gbps support is optional. Can it support 20Gbps but not support 100W charging? Sure, that is optional too. Can my phone's fully-compliant RCS implementation not support upgrading a text message to a video call? Sure! MIVC is optional. Can it fail to send pictures while I am on a call? Sure, that is optional! Can encryption not be supported? You bet, that is optional too! So what does it even mean to comply with the spec then, if everything is optional? It means the spec writers spent too much time engaging in mental masturbation and too little time in contact with the real world, basically. There are two ways this happens: academics who are not aware that outside their offices, there is such a thing as the real world, and design-by-committee situations, where the real world simply never gets a seat at the table -- having failed to file a motion to be seated there in time for the chairman to bring it to a vote.

Just scope out this line: "The RISC-V B (Bit-Manipulation) extension is a standard collection of instruction set enhancements designed to improve performance and code density through efficient bit-level operations. It is split into distinct sub-extensions: Zba, Zbb, Zbc, and Zbs." Only a design-by-committee process would ever unironically produce this sequence of words.

When writing a spec, every single thing you make optional, you split the possible implementations into two incompatible groups. Do this enough times and you end up with your spec being meaningless. And boy, did the designers of RISC-V screw the pooch here. Everything is optional! Multiplication -- optional. Division -- optional. Support for a user mode -- optional. Supervisor mode -- optional. CSRs -- optional. Compressed instructions -- optional. "Extra compressed instructions" -- still optional. I bet that if they thought they could get away with it, they'd make addition optional!

Every single thing you make optional, you split the possible implementations into two incompatible groups.

The basic instruction set of RISC-V, at first publication time, included CSRs, which, as I had mentioned, are required for a standards-compliant method of handling interrupts as well as for support of differing privilege levels. That is not unreasonable; it is sane and *not broken*. Which is, of course, why they fixed it ... ASAP! CSRs got pulled out into an extension called Zicsr, and now the base ISA lacks ability to handle

interrupts or provide privilege separation. But it is actually, and hilariously, much more idiotic than that. Let's say you have a lot of things that are optional. What is the first thing code would want to know? "Is feature X implemented on my hardware?" of course. CPUID is how you answer this question on x86. How do you do it on RISC-V? Well, I have good news and bad news. There is a CSR called misa which can answer some of those questions (not all of course, that would be *too* sane). Did you spot the problem yet? It is a CSR and CSR support is optional (Zicsr extension is not mandatory). If that was not enough of a crotch punch, misa is not required to be accurate if implemented -- it is allowed to read as all zeroes -- it being meaningful is itself entirely optional. Yup... the only way you have to detect optional features is optional. But wait, there is more!

The "M" in front of "misa" indicates that this is a machine-mode CSR. Machine mode is the highest privilege mode in RISC-V (and the only non-optional one, if you're keeping track). This register is not readable from lower-privileged modes, even if you are lucky enough to (a) be running on hardware that implements Zicsr, (b) be running on hardware where misa is not hardwired to be all zeroes, and (c) running on hardware that implements other privilege modes. This means that tailoring your code to the capabilities of the hardware is not possible for normal user code. If your hardware implements the *OPTIONAL* supervisor mode, it also cannot detect the core features. The party line is "ask the machine mode supervisor". The problem, obviously, is that you have no idea what that supervisor is or how to "ask" it. There is a common one in use called OpenSBI, but there is, of course, no way to detect if that is what your machine mode runs.

Another fun bit of optionality here is the system timer. RISC-V spec has a timer; it is optional, of course. It is *not* accessed using CSRs, because of course not! That would be too consistent. The official party ~~line~~excuse is that this was done to conserve the encoding space in the CSRs. I guess this is because they expected to run out of ... 4096 of them⁉ A bit ambitious if you ask me -- no current architecture comes even close, not even x86. But let's move on. If the timer registers are not CSRs then where are they? They are memory mapped! Where? Well, since a timer is a core peripheral that any OS would need, and since the CPU core spec specified it, it, of course, is at a well defined address that you can rely on. Just kidding! Nothing in this spec is that sane! The address is "implementation defined" and can be anywhere at all. Good luck, have fun, don't crash!

I shall tell you of just one more fun situation here, of the many I could: exception and interrupt vectoring. When an exception or an interrupt occurs (assuming the optional Zicsr is implemented), where does the CPU jump? Depending on a lot of optional and optionally-supported config regs, delegation regs, and all sorts of other overcomplicated nonsense, eventually the core will pick to use machine or supervisor

vector register (mtvec or stvec). That CSR points to the handler, except its bottom two bits that determine its "mode". What is a mode? There are two modes documented. The direct mode is when the lower two bits are 0b00, in which case all exception and interrupts just jump to the address in the higher bits. The vectored mode (lower bits 0b01) is meant to simplify and speed up interrupt handling. All exceptions jump to the address in the higher bits of the reg, while all interrupts jump to that address plus 4 times the interrupt number. So what is my problem with this seemingly sane design? That both of the modes are optional!!!! No part of the spec mandates even the simple direct mode! It is possible, at runtime, to detect if a given mode is implemented by writing the lower bits, reading them back, and seeing if they stuck. But it would be entirely valid to implement a core with only vectored mode supported. Or only direct mode supported, or both, or neither, if instead your core vendor invented their own separate mode. This makes writing any sort of a generic kernel very difficult -- you literally have no idea what to expect. Why direct mode was not made mandatory I cannot fathom, but I can surely tell you that the person who decided that wore oversized shoes, had a big red nose, and wore a lot of white face makeup.

It really looks like the authors had heard of Popek & Goldberg, but failed to read past the abstract.

At this point in time you might jump to the defence of this indefensible idiocy by shouting one of two things: "other architectures also have optional features" and/or "hiding misa is needed to support virtualization, haven't you read Popek & Goldberg?". Let's demolish these feeble excuses one at a time. For "other architectures" we'll consider things in common use in the last few decades: x86, ARMv7, and Aarch64. x86, as previously mentioned, has CPUID which will happily tell you which features the current core has. It will do this quite easily in user mode, as one would expect. Arm has ID_AA64PFR0_EL1 and ID_AA64ISAR0_EL1 available to the kernel at least (though not to userspace). But there is a much more important point to be noted here, which explains why ARM's design is not fatal. In both x86 and in ARM, optional features are of two clear classes: (1) high performance compute that is usually programmed using intrinsics or hand-rolled assembly for tight loops in special circumstances (video encoding, fluid simulations) or by libc (memcpy, memset, strlen), and (2) NOP-compatible optional things that can be safely run on hardware that does not support them since it will execute as a NOP and that is safe. For x86 that would be endbr64 and for aarch64 that would be almost all of the PAC instruction set. Note that at no point are things needed in completely normal compiled code optional. Multiplication, division, addressing modes, are always available. This means that a normal C compiler targeting these architectures does not face the impossible choice of: "compile for the lowest possible denominator to allow code to run on all arch versions" or "assume things like multiply and sane addressing modes

exist and prepare to crash on a core that chose not to implement them". Yes, of course, this is where people will say "just target your exact core, why don't you?" Yes, I never said that the idiocy of this ISA cannot be overcome with enough contortion. I said that this sort of poor design was acceptable in the 1970s when we did not know better and is inexcusable in the 2000s, as now we do.

Now, on to the virtualization excuse. First of all, Popek & Goldberg talk about an architecture being virtualizable specifically in the context of it lacking special virtualization support. Indeed by trapping every instruction that acts differently in user and supervisor mode, one can virtualize any architecture. But the alternative is just building-in virtualization support. x86 is not virtualizable as per Popek & Goldberg, at least due to POPF instruction. And yet I have VMs running on my x86 box just fine, as x86 added support for virtualization. It was nontrivially difficult to bolt it on post-facto, but it was done. If doing it at architecture design time, it is trivial. Which is to say that *ANY* mention of Popek & Goldberg to justify decisions made at architecture design time is bullshit. Arch design time is precisely the time to *do it right*. Popek & Goldberg even mention that trapping everything is theoretically interesting for the proof of virtualization but not practical. **NOT PRACTICAL**. So what did the designers of RISC-V do? They justify misa being not exposed to supervisor and user mode with "but virtualization... what if the hypervisor wants to hide capabilities from a VM?". Bull ... let it arrive ... shit! x86 and ARM both manage that just fine. And RISC-V could have too, simply by allowing the hypervisor to lie about misa's contents while letting everyone read it still. It really looks like the authors had heard of Popek & Goldberg, but failed to read past the abstract.

Another thing they excuse by a vague hand wave in the direction of Popek & Goldberg is the inability of the executing code to detect what CPU mode it is in. This is, again, utter nonsense. x86 exposes this indirectly via POPF (for example) and ARM does not even make you trick it, exposing it completely openly in the CurrentEL MSR. Detecting the current mode on RISC-V is an adventure. It is sometimes possible, but not in all cases. Why might you need this? For example, if you are writing a kernel and want it to support all RISC-V cores. I spent a bit of time trying to make this work for my kernel for rePalm, so I can walk you through the decision tree and show where each branch comes to life and whacks you in the gonads. First of all, if the core has no Zicsr, you are guaranteed to be running in machine mode, but, there is no way to know that there is no Zicsr, other than probing by doing a CSR read, but there are two problems: first, without Zicsr, there is no proper generic way to catch the resulting exception when an invalid instruction trap is generated. Second, what CSR to read? Don't forget that likely all of them are optional. One might be tempted to go for misa, but do not forget that you are probing what mode you are in. If you are in supervisor mode, that probe would also fail, even if Zicsr was

implemented. Ok, you can try reading sstatus. That one is readable to supervisor mode AND machine mode. You're safe, right? You wish! Supervisor mode is optional, and if your core does not implement it, there is no sstatus register, so ... you trap. Ok. Let's simplify the problem. Let's assume Zicsr exists. Can you then at least tell apart S mode from M mode? Nope! The following seems like a tempting solution: set stvec to point to your handler that simply adjusts sepc forward by 4 and returns (skipping the faulting instruction), then execute a read of misa. If you were in machine mode, it reads fine. If you were in supervisor mode, it traps, and any sane machine monitor would hand you an illegal instruction trap. Then, your handler would skip the instruction, and you'd note this and conclude you were in supervisor mode. You win, right? Almost... Once again: supervisor mode is optional. Let's imagine you were in machine mode on a core without supervisor mode support. You'd trap as soon as you tried to set stvec, since it does not exist. You might be tempted to say: why not just catch that trap too? Because to do that, you need to set mtvec, and you cannot be sure you can do that since you might have been in supervisor mode all along. Thus the intersection of everything being optional and the authors' complete misunderstanding of Popek & Goldberg lands the poor you in a pile of shite.

Missing Obvious Pieces

On average every other function uses or could use an instruction to branch on the value of a bit.

Despite having an extension for seemingly everything, including operations on the common kitchen sink, somehow a number of obviously-useful instructions are missing. I already covered the lack of register + register addressing modes, so we'll not bother returning to that. There are a few other obvious low-hanging fruit that were seemingly ignored. And before you argue that RV32I was designed to be simple, all of the things I am about to suggest are trivial in the extreme, mostly reducing to simple wires on an ASIC.

First and foremost: test a bit and branch based on it. This one instruction replaces two (SLLI + BGEZ/BLTZ), but also it does not require a temporary register. To check how common this would be, if it existed, I grabbed a random aarch64 binary (the latest raspian kernel for raspberry pi, "vmlinuz-6.1.0-49-arm64", sha256: B3B686DE 82CC7B84 EFEB8F6B 309A4E6C 53E7461F 281D3E56 F42E4AA3 B6207075), disassembled it, and counted the number of instances of TBZ/TBNZ. There were 35,393 instances. By comparison, there are 70,109 instances of RET, which means that on average every other function uses or could use an instruction to branch on the value of a bit. Implementing it is trivial in hardware, and indeed branching on a bit is extremely common in dissecting protocols or using bitfields. For big out-of-order cores, renaming is simpler when one fewer register gets clobbered, and also there is

no need to try to fuse two instructions when one exists. For smaller MCU cores, where no fusion exists, this is a simple code size and speed win. Why this obvious thing was not done, I do not know.

My next major gripe - bitfield operations - bit field extract and bitfield insert. These are extremely useful for working on things like network packets and hardware registers. Bit field extract can be simulated using two instructions - SLLI + SRLI/SRAI based on the desired signedness. Bitfield insert takes a lot more work to simulate: create the inverse mask, AND with destination reg, shift source reg into place, OR into destination. Depending on the bits, it is 3-6 instructions easily. BFC (bit field clear) is a simpler special case that is also quite useful. It is doable in 2-3 instructions. In hardware though, it is just wires - no complex logic, no nothing! There is no need to be clever like aarch64 is, although the designers of RISC-V could learn a lesson or ten from aarch64 indeed, including clever bitfield handling. I did the same counting exercise using the same kernel image. There are 6,284 bitfield insert instructions and 8,881 bitfield extract instructions -- two out of every 9 functions on average use these bitfield ops. To add insult to injury, there *is* a bit-ops extension for RISC-V -- Zbs. By looking at it, you can tell the authors were academics. It is clean, simple, easy to explain, elegant, and completely useless. Who the hell ever needs to extract just one bit? Seriously, what a missed opportunity.

Ridiculous encoding

RISC-V is the first architecture I've ever encountered which scatters immediates randomly throughout the instruction with no immediately-clear reason for it. This makes emulating it a huge pain since it takes so very long to recombobulate the immediate values, compared to architectures like MIPS, or ARM. The former simply uses bits 0..15 for immediates, the latter has fancier encodings, but at least there are just a few. RISC-V designers' justifications for this were "the same bits of the immediate come from the same bits of the instruction" -- a justification so idiotic that it physically hurts to attempt to pretend to believe it. It is the sort of thing that a software person who's never written verilog would think helps make things easier. You see, no matter how you spin it, you will need a mux for immediates, since they are of differing lengths and with differing number of trailing zeroes, depending on the instruction. And that mux, well ... it does not care even a little which instruction bits are wired into its inputs, it is all just wires. But even if the reasoning for "why" is idiotic, let us inspect the claim itself, to see if they did accomplish what they claimed to have wanted. Do the same bits of immediates always come from the same place? Let's take a look. For I-type instructions, bit 1 of the immediate comes from instruction bit 21, bit 11 of the immediate comes from instruction bit 31. For S-type instructions, the same bits of immediates come from instruction bits 8 and 31 respectively. For B-type instructions,

they come from bits 8 and 7 respectively. And for J-type instructions, they come from bits 21 and 20 respectively. As you see, they indeed always come from the same place, as promised, if we merely ignore the meanings of the words "same" and "place".

But this just barely touches the surface of the insanity of the encodings! For J-type instructions, the immediate value is scattered in the following order: 20 10 9 8 7 6 5 4 3 2 1 11 19 18 17 16 15 14 13 12. What possible justification could you imagine for this insanity, other than that the designers confused the chatter of a bingo parlor for the proper bit order for immediates. And yet, this is *nothing* compared to the mess that they made of the compressed instruction set...

There are no fewer than 9 (nine!) instruction formats here, and that does not include Zcb, which adds 8 more! But even that is not all! Depending on the instruction, the same format (eg: CI) encodes immediates differently in the same bit positions. Accounting for all of that, it is almost one instruction encoding format per instruction! It did not need to be like this! Thumb and microMIPS both give examples how to not fuck up this badly, and yet, despite easy availability of examples of how to do it right, RISC-V designers bid us hold their collective LSD-laced beers and went at it in the most pessimal way imaginable. Let us first, of course, look at the immediates in C, since "they always come from the same place in the instruction word", you know ;)

When talking about immediate encodings henceforth, I shall use "x" to indicate when the immediate is broken into pieces and there is something else there in the instruction. L.LWSP (which loads a word from the stack) encodes the immediate in this order: 5 x x x x 4 3 2 7 6, C.SWSP which is its sibling for storing to stack, instead, encodes the immediate as 5 4 3 2 7 6. C.LW (which loads a word from memory addressed by a register) encodes its immediate as: 5 4 3 x x x 2 6, naturally. Its sibling, C.SW uses the same encoding, indicating that the design team missed an opportunity to scramble some bits here. If you wanted to load a byte from memory, you'd use C.LB, whose immediate encoding is, of course, nothing like the above. It uses bit order: 0 1. If you wanted to load a halfword, you'd use C.LHU, whose bit order is just: 1. Because of course it is! I am not even going to touch on CM.PUSH and CM.POP because their encoding is so complex that the spec spends a whole chapter explaining how to decode them. Truly, a sign of a simple and intuitive encoding, if you ask me.

While claiming to make use of all possible encoding space in the 16-bit instruction space, some fruit remain so low-hanging as to require OSHA warnings! A simple example: logical shifts include 6 bits of immediate. The argument is that this is needed for 64-bit instructions, but there are already many encodings in the compressed instruction set that are RV64-only. That is to say that RV64C and RV32C

are *already* incompatible. Given that, why the hell are all shift instructions in RV32C carrying an extra zero bit? You cannot shift a 32-bit register by more than 32 bits. The spec says that bit must be zero, and yet no encoding uses the space opened up by that bit being one. Self delusion is telling yourself you are making good use of encoding space while also carrying around the ability to shift 32-bit registers by 61 bits, my friends. RV32E is even more egregious since many of its compressed instructions carry an extra bit to encode registers that do not exist there. As RV32E is ABI-incompatible with RV32I and they would never share code, giving RV32E more useful encodings by using those extra bits that would have encoded registers x16..x31 would have been an excellent idea. That is probably why it was not done.

Moving on... C.J's bit order could likely pass the NIST Statistical Test Suite for random number generators: 11 4 9 8 10 6 7 3 2 1 5. What even? C.BEQZ/C.BNEZ are also jumps, though conditional, so of course their encodings have almost nothing in common with the previous one, they use: 8 4 3 x x x 7 6 2 1 5. When encoding immediates for C.LI, the bit order is 5 x x x x x 4 3 2 1 0. That almost looks sane, but do not despair, more fun is coming. Let us say you wish to adjust the stack pointer. C.ADDI16SP is here for you, with its immediate encoded as: 9 x x x x 4 6 8 7 5. And if you wanted to get an address of a stack variable, C.ADDI4SPN is there, with its immediate encoded as: 5 4 9 8 7 6 2 3. All very logical, sane, and clear, as promised.

Imagine you are a CPU (or an emulator) trying to decide how to decode an instruction. If you are a sane CPU, most likely it goes like this: look at 1-3 bits to determine instruction format, from there look at 2-5 bits to figure out the instruction, and you are done, you are ready to execute. If you are RISC-V, things are a bit ... more complex. First you look at the bottom 2 bits to figure out if the instr is 2 or 4 bytes, for 2-byte instructions, you look at the top three bits to determine what instruction this is. So far, so good. But then... you notice that this instruction has an immediate. You need to reassemble the jigsaw puzzle that that is. And then you recall that some register-register instructions are encoded in the immediate format, with a magic immediate value indicating that they are register-register ops. For example C.NOT is encoded this way, the magic value being 0b111101. C.ZEXT.W (if your hardware implements the proper mishmash of extensions to have it at all) uses the magic immediate 0b111100. Somehow microMIPS and Thumb managed to do without this insanity. How? Ancient secrets that apparently were irrecoverably lost before the RISC-V authors were born.

If this were not enough, it is also notable that there are conflicting encodings in the compressed instruction set, depending on which extensions are implemented. Some extension combinations are simply impossible (eg: Zcmp and D). This is a bigger fuck-up than all the previous ones since it is not merely cosmetic or efficiency-related. Despite *DECADES* of accumulating backwards-compatibility cruft, even x86 has

managed to avoid the obvious trap that is having the same byte sequence mean different things to different implementations of the same architecture. I repeat: the architecture with the famously-terrible encoding managed to preserve semantic stability across almost 50 years, while the clean-sheet architecture designed by people who had decades of hindsight apparently didn't manage to do it across five years. It is OK for an architecture to have unimplemented encodings that become implemented instructions in later versions. It is justifiable to have implemented instructions that become unimplemented later. However, having encodings change meanings (or worse: start off having different meanings) in different implementations of the same architecture is insane! I think I recall seeing a whole chapter on this in the DSM-5!

Having encodings change meanings [...] in different implementations of the same architecture is insane!

Having this happen means that instead of a clearly-understandable crash you get ... well ... anything. Who can predict how their binary will act when a floating point store silently becomes a double-register move or a jump instruction, or vice-versa? This is not a summary of a plot of a B-side programmer-themed horror flick. It could really happen to you! Consider the instruction 0xA002. Depending on your core, it could store a double-precision floating point register 0 to stack offset 0 (C.FSDSP f0, 0(sp)), or it could jump somewhere (CM.JT 0). But at least a random jump might cause a crash soon enough for you to notice. Consider 0xAC66. Depending on your core, it could store a double-precision floating point register to stack (C.FSDSP f25, 0x18(sp)), or it could move a0 into s0 and a1 into s1 (CM.MVA01S s0, s1). No control flow changes. Just two corrupted registers and a value not stored to stack. Some would attempt to argue that this confusion could never happen, since everyone knows (or should know) what their core is and what it does. Those "some" have clearly never encountered the real world. Oftentimes, you get binary blobs from vendors and you just link them into your microcontroller code. MEMS sensor vendors are notorious for shipping their "super proprietary" calibration algorithms like this. Now, let us imagine that we had such a binary which we linked into shipping code and shipped it on an MCU which had the Zcmp extension. It was all smooth sailing until we upgraded to a larger MCU to have better floating point support for some fancier math our new product needs to do. This new MCU supports double-precision floating point numbers. Suddenly, our binary is crashing randomly in random places and misbehaving. There are no "undefined instruction" traps, no immediately-obvious cause. Turns out, the new core supports the D extension, and thus cannot support Zcmp. But, due to the reused encodings, we did not find out the normal way -- an undefined instruction trap. Instead, we spent weeks debugging random crashes when random registers were getting corrupted. Our disassemblers were no good - they

correctly disassembled the instructions in one of two valid ways. Debuggers were of little help too -- they showed some instructions seemingly doing nothing, with a few words on the stack becoming corrupt soon after. Interrupt handlers were overhauled a few times, wasting a lot of engineer time, to make sure they saved and restored contexts correctly. The MCU vendor's FAEs spent days onsite helping our electrical engineers rule out noise in the power supply lines that could have been causing random register and stack corruptions. Also, the sensor calibration has been broken the entire time since the MCU upgrade, but since sensors usually work plausibly well without calibration and nobody noticed among the random crashes. Now, debugging this, *IS* the plot for a B-side programmer-themed horror flick that I've been shopping around Hollywood for a while. Dear RISC-V committee, you are staying after class today and writing the following on the whiteboard 100 times: upgrading a processor should not turn a valid binary into a semantically different valid binary.

Alleged Fixes

Of all the currently available and recommended at time of publication RISC-V SBCs, approximately none are actually RVA23 compliant. None of them will ever run Ubuntu LTS or future AOSP builds.

Many RISC-V fans will claim that all the optionality is no longer an issue since the RISC-V foundation created "profiles" which is a list of mandatory optional extensions (yes, do read that again) that when implemented together allows one to claim to comply with a profile. Again: the ecosystem has had to invent a second layer of standardization whose entire purpose is to say which parts of the first standard you are actually allowed to assume exist. Really, if you find yourself having to build a second standard whose purpose is to tell everybody which parts of your first standard they must actually implement, perhaps the first standard wasn't quite finished. Only a design-by-committee process could produce this situation. To avoid having to target the lowest common denominator of RISC-V, or the embarrassment of having to build thousands of slightly different builds of every package so as to target every possible extension combination, most vendors who are delusional enough to hope that RISC-V desktops will actually happen are planning to mandate RVA23. Ubuntu, Red Hat, and Android are all in this group. Fun story: of all the currently available and recommended at time of publication RISC-V SBCs, approximately none are actually RVA23 compliant, not StarFive's VisionFive 2, not the Banana Pi BPI-F3, not the Lichee Pi 4A, not the Orange Pi RV2, not even the HiFive Premier P550. None of them will ever run Ubuntu LTS or future AOSP builds. Hilariously, some discussions online have taken to talking about "almost RVA23" cores. Again: the profile designed to solve fragmentation isn't backwards-compatible with much of the hardware that is currently sold as "RISC-V", thus *fragmenting* the RISC-V world into "almost-RVA23"

and post-RVA23, with the majority of the current boards being "almost". The fans will call this profile system a win, but I say it is a direct indictment against this crime of infinite optionality and the people who thought it was ever a good idea. Big desktop-type cores do not need the divide instruction, array addressing, and basic SIMD to be optional. And nobody should have needed *TEN YEARS* to realize this. Additionally, a lot of vendors whose chips will not make the RVA23 cut are likely now screwed, as Android and Linux leave them behind, no matter how "almost" they were. That is what they get for buying into this, I suppose. A lesson to be learned about FOMO and getting in early -- sometimes it only "almost" pays off. As a hilarious side-note, there is one linux distro which is completely and perfectly tuned for this insane RISC-V landscape - Gentoo, since its thing was always: build each package from source to take advantage of the exact host CPU features and optimizations. That is: RISC-V's ideal (read: the only plausible) software distribution model is apparently Gentoo. After I hit "publish" on this article, I shall be off to find myself a contractor to make a Bat-Signal™-style lantern with the Gentoo logo, to aim at the clouds above the RISC-V HQ. Gentoo, you have been summoned to a battle you did not even know you have been preparing for your entire existence!

How Did We Get Here and Where to Now?

If you read the RISC-V rationale/excuse doc, and discard all the parts that we've concluded (in the above sections) to be outright lies, exaggerations, or complete misunderstandings of how existing architectures (and the world at large) work, we end up with this one sentence that likely explains it all: "Ultimately, we thought it was best for our purposes to start from a clean slate, rather than modifying OpenRISC accordingly." This is after they explain that OpenRISC does all they wished to do

except it has delay slots, and proceed to admit that a version without delay slots already existed. There you have it: why the bits are scattered like bingo, why there are hundreds of conflicting extensions with conflicting encodings, and why obviously useful instructions are nowhere to be found -- simply because they all were "**not invented here**".

To be fair, another possible explanation for how we got here is that this is how academia and grants work. By their nature, grant proposals tend to oversimplify and overpromise their real-world applicability and impact. Given that a lot of RISC-V "research" was done under various grants, it is plausible that the incentives of the academic world had an influence. Since this hypothesis is unverifiable with the information available to us, we'll leave it at that.

Does This Mean RISC-V is Doomed?

None of this is to say that RISC-V is doomed. As I said, I fully expect it to take over the space currently occupied by 8051 as a typical cheap embedded core instantiated when some small amount of logic is needed, or some beefy accelerator or DMA engine needs some mild babysitting. Much like the linux kernel -- the price is right. ARM wants licensing fees, while the RISC-V spec is free, and (this part is key) there are cores out there which can be licensed for free. For situations where performance is not a factor and price is, RISC-V will win simply due to its price. "Good-enough" is a low bar in this case, and RISC-V is of the right height to meet it. And look, this market is not glorious, but it matters and it needs new blood. However, it is important that RISC-V not accidentally think it was chosen for being good. It needs to internalize that it was chosen for being cheap. This is not meant as an insult to cheap small cores -- I've written plenty of assembly for all sorts of shitty cores with shittily designed ISAs. RISC-V *is* an improvement over PIC and 8051, as little of a compliment as that is.

'Good-enough' is a low bar in this case, and RISC-V is of the right height to meet it.

For big compute, there are two separate categories, in my opinion. ML accelerators will likely also end up with RISC-V cores attached to them. This is actually pretty close to the above use case, since most of the computation will be done in specialized blocks of silicon, optimized for many many MATMULS per cycle. The RISC-V core will be configuring DMAs and reconfiguring these large blocks to compute the next layer's outputs. A variant of this design might include a RISC-V core with a very wide vector engine bolted to it. This can be used for element-wise operations which are also common in ML workloads. Again, this core need not be fancy or even out-of-order, it will work simply by being wide enough vector-wise. Its integer pipeline will not be a significant factor for its performance, and RISC-V's warts

will be considered the price you pay for not paying to license a Cortex-A55. I expect RISC-V to win this market for the same reason as for small babysitter cores - actual serious per-core CPU perf in the traditional sense is not needed here, so a good-enough core will do. And traditionally, good-enough solutions are always chosen via the "ORDER BY price ASC LIMIT 1;" process.

The second category for big-compute is actual desktops and SBCs that do interactive computation, browsing, gaming, and other such "desktop work". I do not expect RISC-V to be a serious player at the top of this market. Simply put, the architecture is not designed for it, as pointed out above. Additionally, this market has the margins to afford licensing a much-better-designed aarch64 core from ARM, and gain proper support from a much larger corpus of software. Before you get your megaphone to shout about "openness", please note that the openness of the RISC-V spec is not relevant here at all, because an open spec does not magically materialize a well-designed out-of-order core for you for free. And if someone were to design a good out-of-order core, they would not be giving it away for free. An open spec does not mean every implementation is free. The middle and the lower ends of the cheap SBC market will likely remain a mixture of new RISC-V cores and older ARM cores.

© 2012-2026